SpeedReader: Reader Mode Made Fast and Private

Mohammad Ghasemisharif
University of Illinois at Chicago
Chicago, IL, USA
mghas2@uic.edu

Peter Snyder
Brave Software
San Francisco, CA, USA
pes@brave.com

Andrius Aucinas
Brave Software
London, UK
aaucinas@brave.com

Benjamin Livshits
Brave Software & Imperial College London
London, UK
ben@brave.com

ABSTRACT

Most popular web browsers include “reader modes” that improve the user experience by removing un-useful page elements. Reader modes reformat the page to hide elements that are not related to the page’s main content. Such page elements include site navigation, advertising related videos and images, and most JavaScript. The intended end result is that users can enjoy the content they are interested in, without distraction.

In this work, we consider whether the “reader mode” can be widened to also provide performance and privacy improvements. Instead of its use as a post-render feature to clean up the clutter on a page we propose `SpeedReader` as an alternative multistep pipeline that is part of the rendering pipeline. Once the tool decides during the initial phase of a page load that a page is suitable for reader mode use, it directly applies document tree translation *before the page is rendered*. Based on our measurements, we believe that `SpeedReader` can be continuously enabled in order to drastically improve end-user experience, especially on slow mobile connections. Combined with our approach to predicting which pages should be rendered in reader mode with 91% accuracy, `SpeedReader` achieves average speedups and bandwidth reductions of up to 27× and 84×, respectively. We further find that our novel “reader mode” approach brings with it significant privacy improvements to users. Our approach effectively removes all commonly recognized trackers, issues 115 fewer requests to third parties, and interacts with 64 fewer trackers on average, on transformed pages.

CCS CONCEPTS

• **Human-centered computing → Web-based interaction**; • **Information systems → Browsers**; **Clustering and classification**; *Content analysis and feature selection*; • **Security and privacy** → *Privacy protections.*

KEYWORDS

Reader Mode; Boilerplate Removal; Web Document Classification; Web Performance; Ad Blocking

ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3313596

ACM Reference Format:
Mohammad Ghasemisharif, Peter Snyder, Andrius Aucinas, and Benjamin Livshits. 2019. SpeedReader: Reader Mode Made Fast and Private. In *Proceedings of the 2019 World Wide Web Conference (WWW ’19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3308558.3313596

1 INTRODUCTION

“Web bloat” is a colloquial term that describes the trend in websites to accumulate size and visual complexity over time. The phenomena has been measured in many dimensions, including total page size [7], page load time [5, 44, 45], memory needed [30], number of network requests [16, 28], amount of scripts executed [26, 34, 37, 39] and third parties contacted [25, 26, 28]. This work suggests that growth in page size and complexity is outpacing improvements in device hardware. All of this has a predictably negative impact on user experience.

Web users and browser vendors have reacted to this “bloat” in a variety of ways, all partially helpful, but with significant downsides. Ad and tracking blockers are a popular and useful tool for reducing the size complexity of sites. Prior work has shown that these tools can be effective in reducing privacy leaks [31], network use, and extend device memory life. Such tools, which use filter lists, are inherently limited in the scope of improvements they can achieve. While these filter lists are large [42], they are small as a proportion of all URLs on the web. Similarly, while these lists are updated often, they are updated slowly compared to URL updates on the web.

Similarly, “reader mode” tools, provided in many popular browsers and browser extensions, are an effort to reduce the growing visual complexity of websites. Such tools attempt to extract the subset of page content useful to users, and remove advertising, animations, boilerplate code, and other non-core content. Current “reader modes” do not provide the user with resource savings since the referenced resources have already been fetched and rendered. The growth and popularity of such tools suggest they are useful to browser users, looking to address the problem of page clutter and visual “bloat”.

In this work, we propose a novel strategy called **`SpeedReader`** for dealing with resource and bloat on websites. Our technique provides a user experience similar to existing “reader mode” tools, but with network, performance, and privacy improvements that exceed existing ad and tracking blocking tools, on a significant portion of websites. Significantly, `SpeedReader` differs from existing deployed reader mode tools by operating *before page rendering*, which allows it to determine which resources are needed for the page's core content before fetching.

How we achieve speedups. SpeedReader achieves its performance improvements through a two-step pipeline:

(1) SpeedReader uses a classifier to determine whether there is a readable subset of the initial, fetched page HTML. This classifier is trained on a labeled corpus of 2,833 websites (see Section 3), and determines whether a page can be display in reader mode with 91% accuracy.
(2) If the classifier has determined that the page is readable, SpeedReader extracts the readable subset of document *before rendering*, using a variety of heuristics developed in prior research [24] and browser vendors [9, 23], and passes the simplified, reader mode document to the browser's render layer. This tree translation step is described in Section 4.

Deployment. Combined with a highly accurate classifier of "readable" pages, the drastic improvements in performance, reduction in bandwidth use and elimination of trackers in reader mode make the approach practical for continuous use. We therefore propose SpeedReader as a sticky feature that a user can toggle to be always on. This approximates the experience of using an e-book reader, but with strengths of content availability on the web. It is also a suitable strategy for content prerendering or prefetching that could be implemented by web browser vendors, automatically delivering graceful performance degradation in poor connectivity areas or on underpowered mobile devices until the rest of the page content can be fetched for a complete render.

Contributions.

- **Novel approach to Reader Mode** - combining machine-learning driven approach to checking whether content can be transformed to text-focused representation for end-user consumption.
- **Applicability** - we demonstrate that 22.0% of web pages are convertible to reader mode style in a dataset of pages reported popular by Alexa. We further demonstrate that 46.27% of pages shared on social networks are *readable*.
- **Privacy** - we demonstrate that using reader mode in the proposed design provides superior privacy protection, effectively removing all trackers from the tested pages, and dramatically reducing communication with third-parties.
- **Ad Blocking** - we show that our unique reader mode approach blocks ads *at least as well* as existing ad blocking tools, blocking 100% of resources labeled as advertising related by EasyList in a crawl of 91,439 pages, without the need to use hand curated, hard-coded filter lists.
- **Speed** - we find that the lightweight nature of reader mode content results in huge performance gains, with up to 27× page load time speedup on average, together with up to 84× bandwidth and 2.4× memory reduction on average.

Paper organization. The rest of this paper is structured as follows. Section 2 provides background information to place SpeedReader in context. Section 3 describes the design, evaluation and accuracy of the classifying step in the SpeedReader pipeline, and Section 4 gives the design and evaluation of the reader mode extraction step in the SpeedReader pipeline. Section 3.3 measures how many websites user encounters that are amenable to SpeedReader, under several browser use scenarios. Section 5 provides some discussion for how our findings can inform future readability, privacy and performance work, Section 6 places this work in the context of prior research, and Section 7 concludes.

2 BACKGROUND

2.1 Terminology

This subsection presents several terms that are not standardized. We present them up front, to ease the understanding of the rest of the work.

Reader mode. We use the term "reader mode" to describe any tool that attempts to extract a useful subset of a website for a simplified presentation. These tools can be either included in the web browser by the browser vendor, added by users through browser extensions, or provided by third parties as a web service. Our use of the term "reader mode" is generic to the concept, and should not be confused with any specific tool.

Classification and transduction. Reader mode tools generally include both a technique for determining whether a page is readable, which we refer to as "classification", and a strategy for converting the initial HTML tree into a simplified reader mode tree, which we refer to as "tree transduction". Though most reader mode tools include both steps within a single tool or library, they are conceptually distinct.

Readable. We use the term "readable" to describe whether a web page contains a subset of content that would be useful to display in a reader mode presentation. Reader mode presentation works best on pages that are text and image focused, and that are mostly static (i.e. few interactive page elements). Examples of such readable pages include articles, blog posts, and news sites. Reader mode presentation does not work well on websites that are highly interactive, or when a page's structure is significant to the page's content. Examples of such non-readable pages include web applications (e.g. Google Mail, Google Maps) or pages that are indexes of other content.

2.2 Existing Reader Modes

Several popular web browsers include reader modes designed to simplify a page's presentation, so that browser users can read the page's contents without distraction of visual clutter such as advertisements, page animations, and unnecessary page boilerplate (e.g. footers, page navigation, comments).

In this section, we give a brief description of several existing reader mode tools, how they're deployed by their authors, and how they are used in the evaluations given in the rest of this paper.

Readability.js. Readability.js [9] is an open source reader mode library, implemented in JavaScript. It is maintained by Mozilla, and is used for the reader mode function in Firefox. The code is closely related to "Readability" [2], an open sourced library developed by Arc90 and used for their now-defunct readability.com web service. Classification works by looking for the element on the page with the highest density of text and link nodes. If the number and density of text and link nodes in that element exceed a given threshold, the library treats the page as readable. Tree transduction works by normalizing the contents of the text-and-link dense element (to remove styling and other mark up), looking for near-by images for inclusion, and using text patterns in the document that identify the page's author, source and publication date.

Significant to SpeedReader, Readability.js does not consider any display or presentation information when performing either the classification or tree transduction steps. This means that the page does not need to be loaded and rendered to generate a reader mode presentation (though in practice Firefox does not use this library in this way).

Safari Reader View. Safari Reader View is a JavaScript library that implements the reader mode presentation in Safari. Like Readability.js, it is also a fork from Arc90's "Readability", though Apple has changed how the library works in significant ways. In addition to looking for elements with high text and anchor density, Safari Reader View also uses presentation-level heuristics, including where elements appear on the page and what elements are hidden from display by default. Relevant to SpeedReader, this means that Safari Reader View must load a page and at least some of its resources (e.g. images, CSS, JavaScript) to perform either the classification or tree transduction level decisions. Because significant portions of Safari Reader View require a document be fetched and rendered before being evaluated, we do not consider it further in this work (for reasons that are detailed in Sections 3 and 4).

BoilerPipe. BoilerPipe is an academic research project from Kohlschütter *et al.* [24], and is implemented in Java. BoilerPipe has not been deployed directly by any browser vendor. BoilerPipe does not provide functionality for (readability) classification, and assumes that any HTML document contains a readable subset. For tree transduction, BoilerPipe considers number of words and link density features. Like Readability.js, it does not require a browser to load and render a page in order to do reader mode extraction. Their analysis reveals a strong correlation between short text and boilerplate, as well as long text and actual content text (of the textual content) on the Web. Using features with low calculation cost such as number of words enables BoilerPipe to lower the overhead while maintaining high accuracy.

DOM Distiller. DOM Distiller is a JavaScript and C++ library maintained by Google, and used to implement reader mode in recent versions of Chrome. The project is based on BoilerPipe, though has been significantly changed by Google. The classification step in DOM Distiller uses a classifier based approach, and considers features such as whether the page's URL contains certain keywords (e.g. "forum", ".php", "index"), if the page's markup contains Facebook open graph, Google AMP, identifiers, or the number of "/" characters used in the URL's path, in addition to the text-and-link density measures used by Readability.js. At a high level, the tree transduction step also looks at text-and-link dense element in the page, as well as special-cased embedded elements, such as YouTube or Vimeo videos.

DOM Distiller considers some render-level information in both the classification and tree transduction steps. For example, any elements that are hidden from display are not included in the text-and-link density measurements. These render-level checks are a small part of DOM Distiller's strategy. We modified DOM Distiller to remove these display level checks, so that DOM Distiller could be applied to prerendered HTML documents. We note that the evaluation of DOM Distiller in this work uses this modified version of DOM Distiller, and not the version that Google ships with Chrome. We expect this modification to have minimal effects on the discussed measurements, but draw the reader's attention to this change for completeness.

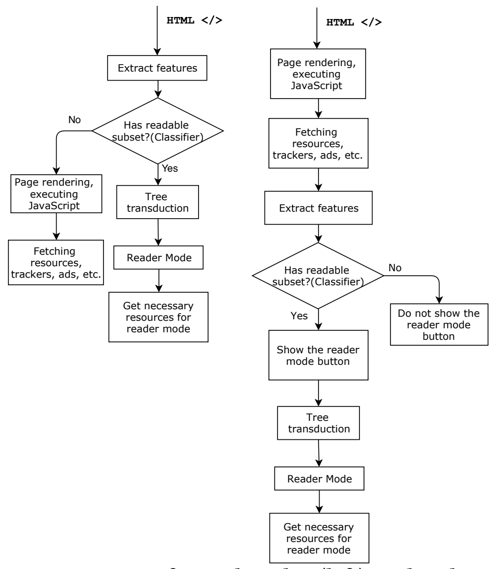
HTML </>
Extract features
Has readable subset?(Classifier)
No
Yes
Page rendering, executing JavaScript
Fetching resources, trackers, ads, etc.
Tree transduction
Reader Mode
Get necessary resources for reader mode
HTML </>
Page rendering, executing JavaScript
Fetching resources, trackers, ads, etc.
Extract features
Has readable subset?(Classifier)
No
Yes
Do not show the reader mode button
Show the reader mode button
Tree transduction
Reader Mode
Get necessary resources for reader mode

Figure 1: Comparison of SpeedReader (left) with other existing reader modes (right)

2.3 Comparison to SpeedReader

The reader mode functionality shipped with all current major browsers is applied after the document is fully fetched and rendered.[1] This greatly restricts the possible performance, network and privacy improvements existing reader modes can achieve. In fact, in some reader mode implementations we measured, using reader modes *increased* the amount of network used, as some resources were fetched twice, i.e. once for the initial page loading, and then again when presenting images in the reader mode display.

Most significantly, SpeedReader differs from existing reader mode techniques in that it is implemented strictly before the display, rendering, and resource-fetching steps in the browser. SpeedReader can therefore be thought of as a function that sits between the browser's network layer (i.e. takes as input the initial HTML document), and returns either the received HTML (when there is no readable subset), or a greatly simplified HTML document, representing the reader mode presentation (when there is a readable subset). Figure 1 provides a high level comparison of how SpeedReader functions, compared to existing reader modes.

The fact that SpeedReader only considers features available in the initial HTML and URL enables SpeedReader to achieve performance orders of magnitude above existing approaches. Figure 2

[1]While Readability.js does not require that the page be rendered before making reader mode evaluations, in practice Firefox does not expose reader mode functionality to users until after the page is fetched and loaded.

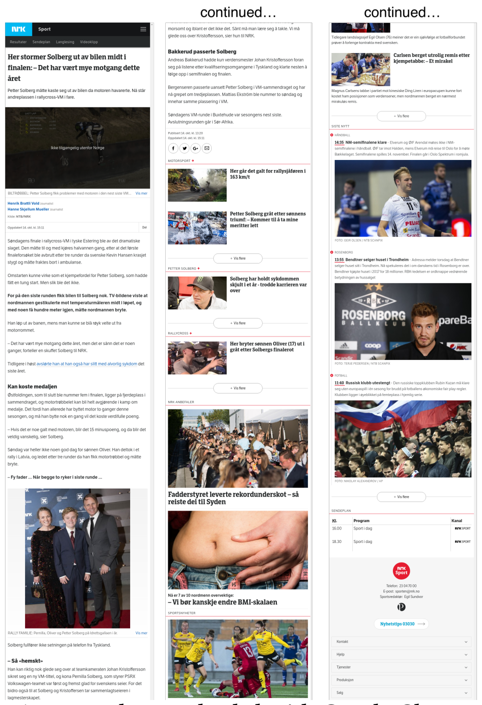
continued…
continued…
Her stormer Solberg ut av bilen midt i finalen: – Det har vært mye motgang dette året
Kan koste medaljen
Faddersttyret leverte rekordunderskot – så reiste dei til Syden
– Vi bør kanskje endre BMI-skalaen
ROSENBORG

Figure 2: An example page loaded with Google Chrome browser with no modifications

provides a strawman example of a news page as delivered to a standard client: including portal branding and content, but also a range of links to different articles, images and trackers, for a total of 2.7MB of data transferred and 53 scripts executed. Figure 3 demonstrates the functionality of SpeedReader when applying existing reader mode transducers to just the initial HTML document. Therefore, for documents SpeedReader determines are readable, the sources of SpeedReader improvements include:

- Never fetching or executing script or CSS.
- Fetching far fewer images or videos (since images and videos not core to the page's presentation are never retrieved).
- Performing network requests to far fewer third parties (zero, in the common case).
- Saving processing power from not rendering animations, videos or complex layout operations, since reader mode presentations of page content are generally simple.

The above are just some of the ways that SpeedReader is able to achieve considerable performance improvements. The following sections describe how SpeedReader's classification and tree transduction steps were designed and evaluated, and what percentage of websites are amenable to SpeedReader's approach.

3 PAGE CLASSIFICATION

SpeedReader uses a two stage pipeline for generating reader mode versions of websites. This section presents the design and evaluation of the first half of the pipeline, the classification step.

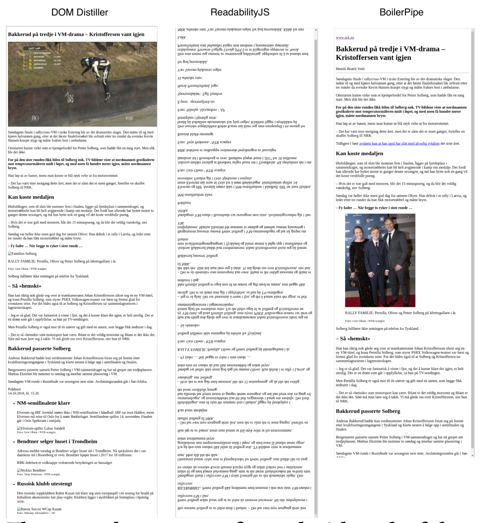
DOM Distiller
ReadabilityJS
BoilerPipe
Bakkerud på tredje i VM-drama – Kristoffersen vant igjen
Kan koste medaljen
Bakkerud passerte Solberg

Figure 3: The example page transformed with each of the evaluated SpeedReader transducers

3.1 Classifier Design

The classification step of SpeedReader uses a random forest classifier, trained on a hand-labeled data set of 2,833 websites. Our classifier takes as input a string, depicting an HTML document, and returns a boolean label of whether there is a readable subset of the document. We note that the input to the classifier is the initial HTML returned by the server, and not the final state of the website after JavaScript execution.

Our classifier is designed to execute quickly, since document rendering is delayed during classification. The classifier is trained using 50 estimators, it expands the nodes until all leaves are pure or contain less than 2 samples, and considers 21 features, each selected to be extractable quickly. Selected features include the number of text nodes, number of words, the presence of Facebook open graph or Google AMP markup, and counts for a variety of other tags.

Our classifier considers the following features. We have made the source code for our classifier available publicly as well.[2]

- Counts of the following tags: `<p>`, `<ul>`, `<ol>`, `<dl>`, `<div>`, `<pre>`, `<table>`, `<select>`, `<article>`, `<section>`, `<blockquote>`, `<a>`, `<img>`, `<script>`
- Count of block elements that contain at least 400 words.
- # of words in block elements that match above condition.
- Number of path segments in the URL.
- Boolean determination if the page has any of the following metatags: `amphtml`, `fb_pages`, `og_article`.
- Boolean determination if the page has plaintext match for any of `schema.org` markup for `Article`, `NewsArticle` or `APIReference`.

3.2 Classifier Accuracy

The goal of the classifier in SpeedReader is to predict whether the end result of a page's fetching and execution will result in a readable page, based on the initial HTML of the page. This section describes the data set we used to both train the SpeedReader classifier, and

[2]https://github.com/brave/speedreader-paper-materials.git

Table 1: Description of data set used for evaluating and training "readability" classifiers.

Data set	Number of pages	% Readable
Article pages	956	91.8%
Landing pages	932	1.5%
Random pages	945	22.0%
Total	**2,833**	**38.8%**

Table 2: Accuracy measurements for three classifiers attempting to replicate the manual labels described in Table 1.

Classifier	Precision	Recall
ReadabilityJS	68%	85%
DOM Distiller	90%	75%
SpeedReader Classifier	**91%**	**87%**

to evaluate its accuracy against existing popular, deployed reader mode tools.

Data Set. To assess the accuracy of our classifier, we first gathered 2,833 websites, summarized in Table 1. Our data set is made up of three smaller sets of crawled data, each containing 1,000 URLs, each meant to focus on a different kind of page, with a different expected distribution of readability. 1,000 pages were URLs selected from the RSS feeds of popular news sites (e.g. The New York Times, ArsTechnica), which we expected to be frequently readable. The second 1,000 pages were the landing pages from the Alexa 1K, which we expected to rarely be readable. The final 1,000 pages were selected randomly from non-landing pages linked from the landing pages of the Alexa 5K, which we expected to be occasionally readable. We built a crawler that, given a URL, recorded both the initial HTML response, and a screenshot of the final rendered page (i.e. after all resources had been fetched and rendered, and after JavaScript had executed). We applied our crawler to each of the 3,000 selected URLs. 167 pages did not respond to our crawler, accounting for the difference between the 3,000 selected URLs and the 2,833 pages in our data set.

Finally, we manually considered each of the final page screenshots, and gave each a boolean label of whether there was a subset of page content that was readable. We considered a page readable if it met the following criteria:

(1) The primary utility of the page was its text and image content (i.e. not interactive functionality).
(2) The page contained a subset of content that was useful, without being sensitive to its placement on the page.
(3) The usefulness of the page's content was not dependent on its specific presentation or layout on the website.

This meant that single page applications, index pages, and pages with complex layout were generally labeled as not-readable, while pages with generally static content, and lots of text and content-depicting media, were generally labeled readable. We also share our labeled data,[3] and a guide to the meaning behind the labels,[4] to make our results transparent and reproducible.

Evaluation. We evaluated our classifier on our hand labeled corpus of 2,833 websites, performing a standard ten-fold cross-validation.

[3]https://github.com/brave/speedreader-paper-materials/blob/master/labels.csv
[4]https://github.com/brave/speedreader-paper-materials/blob/master/labels-legend.txt

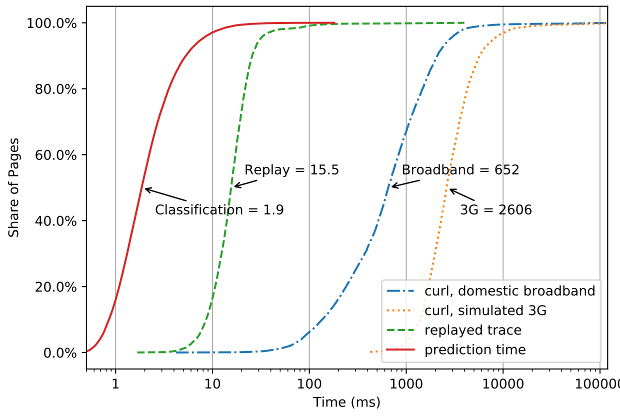

Figure 4: Time to fetch initial HTML document.

For comparison sake, we also evaluated the accuracy of the classification functionality in Readability.js and our modified version of DOM Distiller when applied to the same data set, to judge their ability to predict the final readability state of each document, given its initial HTML. We note that Readability.js is designed to be used this way, but that this prediction point is slightly different than how DOM Distiller is deployed in Chrome. In Chrome, DOM Distiller labels a page as readable based on its final rendered state. This evaluation of DOM Distiller's classification capabilities should therefore not be seen as an evaluation of DOM Distiller's overall quality, but only its ability to achieve the kinds of optimizations sought by `SpeedReader`. Table 2 presents the results of this measurement. As the table shows, `SpeedReader` strictly outperforms the classification capabilities of both DOM Distiller and Readability.js. DOM Distiller has a higher false positive rate than our classifier, while Readability.js has a higher false negative rate.

3.3 Classifier Usability

Problem Statement. Our classifier operates on complete HTML documents, before they are rendered. As a result, the browser is not able to render the document until the entire initial HTML document is fetched. This is different from how current browsers operate, where websites are progressively rendered as each segment of the HTML document is received and parsed. This entails a trade off between rendering delay (since rendering is delayed until the initial HTML document) and network and device resource use (since, when a page is classified as readable, far fewer resources will be fetched and processed).

In this sub-section, we evaluate the rendering delay caused by our classifier, under several representative network conditions. The rendering delay is equal to the time to fetch the entire initial HTML document. We find that the rendering delay imposed is small, especially compared to the dramatic performance improvements delivered when a page is readable (discussed in more detail in Section 4).

Classification Time. We evaluated the rendering delay imposed by our classifier by measuring the time taken to fetch the initial HTML for a page, under different network conditions, and compared it against the time taken for document classification.

Table 3: Measurements of how applicable our readability strategy is under common browser use scenarios.

Measurement	# measured	# readable	% readable
Popular pages	42,986	9,653	22.5%
Unpopular pages	40,908	8,794	21.5%
Total: Random crawl	**83,894**	**18,457**	**22.0%**
Reddit linked	3,035	1,260	41.51%
Twitter linked	494	276	31.2%
RSS linked	506	331	65%
Total: OSN	**4,035**	**1,867**	**46.27%**

First, we determined how long our classifier took to determine if a parsed HTML document was readable. We did so by parsing each HTML string with *myhtml*, a fast, open source, C++ HTML parser [4]. We then measured the execution time taken to extract the relevant features from the document, and to return the predicted label. Our classifier took 2.8 ms on average and 1.9 ms in the median case. Next, we measured the fixed, simulation cost time of serving each web page from a locally hosted web server, which allowed us to account for the fixed overhead in establishing the network connection, and similar unrelated browser book keeping operations. This time was 22.3 ms on average, and 15.5 ms median.

Finally, we selected two network environments to represent different network conditions and device capabilities web users are likely to encounter: a fast, domestic broadband link, with 50 Mbps uplink/downlink bandwidth and 2 ms latency as indicated by a popular network speed testing utility,[5] and a simulated 3G network, created using the operating system's *Network Link Conditioner*.[6] We use a default 3G preset with 780 kbps downlink, 330 kbps uplink, 100 ms packet delay in either direction and no additional packet loss. Downloading the documents on such connection took 1,372 ms / 652 ms (average/median) and 4,023 ms / 2,516 ms for the two cases respectively. Figure 4 summarizes the results of those measurements. Overall, the approximately 2.8 ms taken for an average document classification is a tiny cost compared to just the initial HTML download on reasonably fast connections. It could potentially be further optimized by classifying earlier, i.e. when only a chunk of the initial document is available. Initial tests show promising results, however this adds significant complexity to patching the rendering pipeline and we leave it for future work.

3.4 Applicability to the Web

While subsequent sections will demonstrate the significant performance and privacy improvements provided by SpeedReader, these improvements are only available on a certain type of web document, those that have readable subsets. The performance improvements possible through SpeedReader are therefore bounded by the amount of websites users visit that are readable.

In this subsection, we determine how much of the web is amenable to SpeedReader, by applying our classifier to a sampling of websites, representing different common browsing scenarios. Doing so allows us to estimate the benefits SpeedReader can deliver as well as its relevance to the web. As presented in Table 3, we find that a significant number of visited URLs are readable, suggesting that SpeedReader can deliver significant privacy and performance improvements to users. This subsection continues by describing how we selected URLs in each browsing scenario.

Websites by popularity. We first estimated how many pages hosted on popular and unpopular domains are readable. To do so, we first created two sets of domains, a popular set, consisting of the most popular 5,000 domains, as determined by Alexa, and an unpopular set, comprising a random sample of pages ranked 5,001–100,000. For each domain, we conducted a breadth three, depth three crawl. We first visited the landing page for the domain, and recorded all URLs linked to pages with the same TLD+1 domain. Then we selected up to three URLs from this set, and repeated the above process another time, giving a maximum of 13 URLs per domain, and a total data set of 91,439 pages. The crawl was conducted from AWS IP addresses on 17-20 October 2018.

Social network shared content. We next estimated how much content linked to from online social networks is readable, to simulate a user that spends most of their browsing time on popular online social networks, and generally only browses away to view shared content. We gathered URLs shared from Reddit and Twitter. We gathered links shared on Reddit by using RedditList [32] to obtain top 125 subreddits ranked based on their number of subscribers. We then visited the 25 posts of each popular subreddit and extracted any shared URLs. For Twitter, we extracted shared links from the top 10 worldwide Twitter trends by crawling and extracting shared links from their Tweets.

RSS / feed readers. Finally, we estimated how much content shared from RSS feeds is readable, to simulate a user who finds content mainly through an RSS (or similar) aggregation service. We built a list of RSS-shared content by crawling the Alexa 1K, identifying websites that included RSS feeds, and fetching the five most recent pages of content in each RSS feed.

3.5 Conclusion

In this section we have described how SpeedReader determines whether a page should be rendered in reader mode, based on its initial HTML. We find that SpeedReader outperforms the classification capabilities of existing, deployed reader mode tools. We also find that the overhead imposed by our classification strategy is small and acceptable in most cases, and dwarfed by the performance improvements delivered by SpeedReader, for cases when a page is judged readable.

4 PAGE TREE TRANSDUCTION

This section describes how SpeedReader generates a reader mode presentation of a page, for pages that have been classified as readable. Our evaluation includes three possible reader mode renderings, each presenting a different trade off between amount of media included, performance and privacy improvements.

Generating a reader mode presentation of an HTML document can be thought of as translating one tree structure to another: taking the document represented by the page's initial HTML and generating the document containing a simplified reader mode version. This process of tree mapping is generally known as tree transduction.

[5]speedtest.net - web service that provides analysis of Internet access performance metrics, such as connection data rate and latency
[6]Network Link Conditioner is a tool released by Apple with hardware IO Tools for XCode developer tools to simulate different connection bandwidth, latency and packet loss rates

Table 4: Description of data set used for evaluating the performance implications of different content extraction strategies.

Measurement	Value
Measurement date	17-20 October 2018
# crawled domains	10,000
# crawled pages	91,439
# domains with readable pages	4,931
# readable pages	19,765
% readable pages	**21.62%**

We evaluate Tree transduction by comparing the performance and privacy improvements of the three techniques (Readability.js, DOM Distiller and BoilerPipe) described in detail in Section 2.2.

4.1 Limitations and Bounds

We note that we did not attempt any evaluation of how users perceive or enjoy the reader mode versions of pages rendered by each considered technique. We made this decision for several reasons.

First, two of the techniques (Readability.js and DOM Distiller) are developed and deployed by large browser vendors, with millions or billions of users. We assume that these large companies have conducted their own usability evaluation of their reader mode techniques, and found them satisfactory to users.

Second, the third considered tree transduction technique, Kohlschütter et al's BoilerPipe [24], is an academic work that includes its own evaluation, showing that the technique can successfully extract useful contents from HTML documents. We assume that the authors' evaluation is comprehensive and sufficient, and that their technique can successfully render pages in reader mode presentations. Finally, we are planning to deploy a tree transducer different from existing techniques and a more thorough subjective evaluation of its presentation is left for future study.

4.2 Evaluation Methodology

We compared the performance and privacy improvements achieved through `SpeedReader`'s novel application of three tree transduction techniques: Readability.js, DOM Distiller and BoilerPipe. We conducted this evaluation in three stages.

First, we fetched the HTML of each URL in the random crawl data set outlined in Table 3, again from an AWS IP. The HTML considered here is only the initial HTML response, not the state of the document after script execution. We evaluated whether each of the 91,439 fetched pages that were classified as readable, by applying the `SpeedReader` classifier to each page. We then reduced the data set to the 19,765 pages (21.62%) were readable.

Second, we revisited each URL classified as readable to collect a complete version of the page. To minimize variations in page performance and content during the testing, we collected the "replay archive" for each page using the "Web Page Replay" (WPR) [22] performance tool. WPR is used in Chrome's testing framework for benchmarking purposes and works as a proxy that records network requests or responds to them instead of letting them through to the source depending on whether it works in "record" or "replay" mode.

Finally, we applied each of the three tree transduction techniques to the remaining 19,765 HTML documents, and compared the network, resource use, and privacy characteristics of each transformed page, against the full version of each page. We evaluate performance and privacy characteristics of each page by visiting the URL as replayed from its archive. These findings are described in detail in the next subsections.

We note that using a replay proxy with a snapshot of content often underestimates the costs of a page load. Despite taking care to mitigate the effects of non-determinism by injecting a small script that overrides date functions to use a fixed date and random number generator functions to use a fixed seed and produce a predictable sequence of numbers, it cannot account for all sources of non-determinism. For all requests that the proxy cannot match, it responds with a `Not Found` response. We notice that it results in a small number of requests being missed, primarily those responsible for dynamic ad loading or tracking. It also occasionally interferes with site publisher's custom resource fetching retry logic, where the same request is retried a number of times unsuccessfully, before the entire page load times out and the measurement is omitted.

Table 5: Performance comparisons of three popular readability tree transducer strategies, as applied to the data set described in Table 4. Values are given as Average, Median. Gain multiplier (×) is calculated for each page load and Average and Median values are reported.

Transducer	Resources (#)		Data (KB)		Memory (MB)		Load Time (ms)	
-	A	M	A	M	A	M	A	M
Default	144	91	2,283	1,461	197	174	1,813	1,069
ReadabilityJS	5	2	186	61	85	79	583	68
Dom Distiller	5	2	186	61	84	79	550	63
BoilerPipe	2	2	101	61	81	77	545	44
Gain (×)								
ReadabilityJS	51	28	84	24	2.4	2.1	20	11
Dom Distiller	52	32	84	24	2.4	2.1	21	12
BoilerPipe	77	48	84	24	2.4	2.1	27	15

4.3 Results: Performance

We measured four performance metrics: number of resources requested, amount of data fetched, memory used and page load time. These results are summarized in Table 5 and Figure 5.

We ran all measurements on AWS `m5.large` EC2 instances. For performance measurements, one test was executed at a time, per instance. For each evaluation, we fetched the page from a previously collected record-replay archive, with performance tracing enabled. Once the page was loaded and the performance metrics are recorded, we closed the browser and proxy, and started the next test. No further steps were taken to minimize the likelihood of test VM performance being impacted by interfering workloads on the underlying hardware. For all tests, we used an unmodified Google Chrome browser, version 70.0.3538.67, rendered in Xvfb.[7] Although profiling has overheads of its own [33], in particular for memory use and load times, we used a consistent measurement strategy across all tests, and therefore expect the impact to also be consistent and minor compared to relative performance gains.

We measured a page's load time as the difference between `navigationStart` and `loadEventEnd` events [46] in the main frame (i.e. the time until all sub-resources have been downloaded

[7]While Chrome "headless" mode is available, it effectively employs a different page rendering pipeline with different load time characteristics and memory footprint.

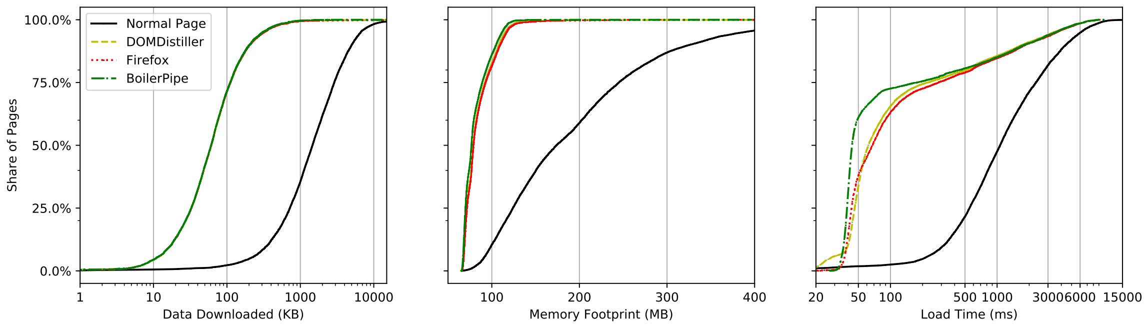

Figure 5: Performance characteristics of the different tree transducer strategies applied, showing the distribution of the key performance metrics.

and the page is fully rendered). Since page content is replayed from a local proxy, network bandwidth and latency variation impact is minimized and the reported load time is a very optimistic figure, especially for bigger pages with more sub-resources as illustrated in Figure 4. Although network cost is still non-zero, the number primarily reflects the time taken to process and render the entire page.

We also recorded the number of resources fetched and the amount of data downloaded during each test. Note that the amount of data downloaded for all of the tree transduction strategies reflects the size of the initial HTML rather than that of the transformed document, as the transformation happens on the client and does not result in additional network traffic. All measured transducers discard the majority of page content (both in page content like text and markup, but also referenced content like images, video files, and JavaScript). Figures 2 and 3 provide an example of how tree transduction techniques simplify page content.

For memory consumption, we measure the overall memory used by the browser and its subprocesses. Google Chrome uses a multi-process model, where each tab and frame may run in a separate process and content of each page also affects what runs in the main browser process. We note that our testing scenario does not consider the case of multiple pages open simultaneously in the same browsing session, as some of the resources are reused. The reported number is therefore that of the entire browser rather than the specific page alone, with some fixed browser runtime overheads.

Memory snapshots are collected with an explicit trigger after the page load is complete with `disabled-by-default-memory-infra` tracing category enabled. Despite including a level of fixed browser memory costs, we still see average memory reduction of up to 2.4× in average or median cases. Overall, depending on the chosen transducer, we show:

- average speedups ranging from 20× to 27×
- average bandwidth savings on the order of 84×
- number of requests is reduced 51× to 77×
- average memory reduction of 2.4×

4.4 Results: Privacy

`SpeedReader` achieves substantial privacy improvements, because it applies the tree transduction step *before* rendering the document, and thus before any requests to third parties have been initiated. The privacy improvements gained by `SpeedReader` are threefold: a reduction in third party requests, a reduction in script execution (an often necessary, though not sufficient, part of fingerprinting online), and a complete elimination of ad and tracking related requests (as labeled by EasyList and EasyPrivacy). This last measure is particularly important, since *92.8%* of the 19,765 readable pages in our data set loaded resources labeled as advertising or tracking related by EasyList and EasyPrivacy [10, 11].

This subsection proceeds by both describing how we measured the privacy improvements provided by `SpeedReader`, and the results of that measurement. These findings are presented in Table 6.

We measured the privacy gains provided by `SpeedReader` by first generating reader mode versions of each of the 19,765 readable URLs in our dataset, and counting the number of third parties, script resources, and ad and tracking resources in each generated reader mode page. We determined the number of ad and tracking resources by applying EasyList and EasyPrivacy with an open-source ad-block Node library [21] to each resource URL included in the page. We then compared these measurements to the number of third-parties, script units, and ad and tracking resource requests made in the typical, non-reader mode rendering of each URL.

We found that all three of the evaluated tree transduction techniques dramatically reduced the number of third parties communicated with, and removed all script execution and ad and tracking resource requests from the page. Put differently, `SpeedReader` is able to achieve privacy improvements at least as good, and almost certainly exceeding existing ad and tracking blockers, on readable pages. This claim is based on the observation that ad and tracking blockers do not achieve the same significant reduction in third party communication and script execution as `SpeedReader` achieves.

Table 6: Comparisons of the privacy implications of three popular readability tree transducer strategies, as applied to the data set described in Table 4. Values are given as Average and Median values.

Transducer	# third-party		# scripts		Ads & Trackers	
	Avg	Med	Avg	Med	Avg	Med
Default	117	63	83	51	63	24
ReadabilityJS	3	1	0	0	0	0
Dom Distiller	3	1	0	0	0	0
BoilerPipe	1	1	0	0	0	0

5 DISCUSSION AND FUTURE WORK

5.1 Reader Mode as a Content Blocker

Most existing reader mode tools function to improve the presentation of page content for readers, by removing distracting content and reformatting text for the browser user's benefit. While the popularity of existing reader modes suggest that this is a beneficial use case, the findings in this work suggest an additional use case for reader modes, blocking advertising and tracking related content.

As discussed in Section 4.4, `SpeedReader` prevents all ad and tracking related content from being fetched and rendered, as identified by EasyList and EasyPrivacy (Table 6). `SpeedReader` also loads between 51 and 77 times fewer resources than typical page rendering and reader modes (Table 5), a non-trivial number of which are likely also ad and tracking related. `SpeedReader` differs fundamentally from existing content blocking strategies. Existing popular tools, like uBlock Origin[20] and AdBlock Plus[15], aim to identify *malicious* or undesirable content, and prevent it from being loaded or displayed; all unlabeled content is treated as desirable and loaded as normal. `SpeedReader`, and (at last conceptually) reader modes in general, take the opposite approach. Reader modes try to identify *desirable* content, and treat all other page content as undesirable, or, at least, unneeded.

Our results suggest that the reader mode technique can achieve ad and tracking blocking quality *at least* as well as existing content blocking tools, but with dramatic performance improvements. We expect that `SpeedReader` actually outperforms content blocking tools (as content blockers suffer from false-negative problems, for a variety of reasons), but lack a ground truth basis to evaluate this claim further. We suggest evaluating the content blocking capabilities of reader mode-like tools as a compelling area for future work.

5.2 Comparison to Alternatives

`SpeedReader` exists among other systems that aim to improve the user experience of viewing content on the web. While a full evaluation of these systems is beyond the scope of this work (mainly because the compared systems have different goals and place different restrictions on users), we note them here for completeness.

AMP. Accelerated Mobile Pages (AMP)[17] is a system developed by Google that improves website performance, in a number of ways. Website authors opt-in to the AMP system by limiting their content to a subset of HTML, JavaScript and CSS functionality, which allows for optimized loading and execution. AMP pages are also served from Google's servers, which provide network level improvements. AMP differs from `SpeedReader` and other reader mode systems in that users only achieve performance improvements when site authors design their pages for AMP; AMP offers no improvement on existing, traditional websites.

Server-Assisted Rendering. Other browser vendors attempt to improve the user experience by moving page, loading, rendering execution from the client to a server. The client then fetches a rendered version of the page from the server (generally either rendered HTML or as a bitmap). The most popular such system is likely Amazon Silk[1]. While there are significant performance upsides with this thin-client technique, they come with significant downsides too. First, user privacy is harmed, since the rendering-server must manage and observe all client secrets when interacting with the destination server on the client's behalf. Additionally, while the server may be able to improve the loading and rendering of the page, its limited in the kinds of performance improvements it can achieve. Server assisted rendering does not provide any of the presentation simplification or content blocking benefits provided by `SpeedReader`.

5.3 `SpeedReader` Deployment Strategies

Always On. `SpeedReader` as described in this work is designed to be "always on", attempting to provide a readable presentation of every page fetched. Although Safari Reader View also supports an "always on" functionality, it lacks performance and privacy enhancement provided by `SpeedReader` (Section 2). While this decision maximizes the amount of privacy and performance improvements provided, it entails an overhead while loading each page (Figure 4), which may not be worthwhile in some browsing patterns such as interacting with application-like sites. Additionally, there may be times when users want to maintain a page's interactive functionality (e.g. JavaScript), even when `SpeedReader` has determined that the page is readable. Ensuring the user's ability to disable `SpeedReader` would be important in such cases. The system described in this work does not preclude such an option, but only imagines changing the default page loading behavior.[8]

Tree Transduction Improvements. The three evaluated techniques in Section 4, which are adapted from existing tools and research, can provide a reader mode presentation with different performance and privacy improvements. Users of `SpeedReader` could select which tree transduction technique best suited their needs. However, we expect that ML and similar techniques could be applied to the tree transduction problem, to provide a reader mode presentation that exceeds existing techniques. An improved tree transduction algorithm would achieve equal or greater performance and privacy improvements, while doing a better job of maintaining the meaning and information of the extracted content. We are currently exploring several options in this area, but have found the problem large enough to constitute its own unique work.

6 RELATED WORK

Content Extraction. The problem of removing boilerplate and extracting relevant content from a webpage has been extensively studied. Previous approaches primarily focused on the code structure, visual representation and the link between the two. Lin *et al.* [29] proposed a method to detect content blocks using `<TABLE>` tags and calculate their entropy to distinguish the informative blocks from the redundant ones. Laber *et al.* [27] proposed a heuristic method for extracting textual sections and title from news articles using `<a>`, `<p>` and `<title>` tags. Other studies have tried to detect *useful* segments in a web page using structural and positional information. Gupta *et al.* [18] introduced a DOM-based method to modify and remove irrelevant DOM nodes to extract the main content. Their

[8]Current browsers and reader modes load all pages in the standard manner, and allow the users to enable a reader mode presentation, while `SpeedReader` would load pages in the optimized reader mode presentation by default, when possible, and allow users to enable the standard loading behavior.

approach utilized filters to remove DOM nodes with advertisements, and link and text ratio thresholds to remove unwanted table cells. While the proposed rule-based method was simple, it had a poor performance in link rich pages where the main content contained many links. Weninger *et al.* [47] introduced a fast algorithm which calculated the HTML tag ratio of each line to cluster and extract text content. Their algorithm did not perform well on home pages as well as it suffered from high recall and low precision. Cai *et al.* [8] introduced a tag-free vision-based page segmentation algorithm to segment a webpage and extract its web content structure using the link between the visual layout and the content. Fan *et al.* [13] introduced Article Clipper, a web content extractor that leveraged visual cues in addition to HTML tags to extract non-textual and textual content and detect multi-page articles. Their approach underperformed in extracting captions which were links as well as images and captions that were outside of main content.

Heuristic methods are limited by their lack of adaptability. Some have proposed learning based methods to overcome this rigidness. Pasternack and Roth [35] described a semi-supervised algorithm, Maximum Subsequence Segmentation, which tokenized HTML into list of tags, words and symbols, and attempted to classify each block as either "in article" or "out of article" text. Kohlschütter *et al.* [24] developed BoilerPipe to classify text elements using both structural and text features, such as average word length and average sentence length. Sun *et al.* [40] proposed Content Extraction via Text Density (CETD) to extract the text content from pages using a variety of text density measurements. Their method relied on the observation that the amount of text in content sections is large, and the text in boilerplate sections contains more noise and links. Sluban and Grčar [38] introduced an unsupervised and language-independent method for extracting content from streams of HTML pages, by detecting commonalities in page structure. While their method outperformed other open-source content extractor algorithms, it suffered from high memory consumption and poor performance in diverse and small HTML data set.

Wu *et al.* [48] proposed a machine learning model using DOM tree node features such as position, area, font, text and tag properties to select and group content related nodes and their children. In their recent paper, Vogels *et al.* [43] presented an algorithm combining a hidden markov model and a convolutional neural networks (CNNs). Their model first preprocessed an HTML page into a Collapsed DOM (CDOM) tree where each single child parent node was merged with its child. CDOM was then segmented into blocks of main content and boilerplate using sequence labeling of DOM leaves. The features were then used to train two CNNs, obtain potentials and finally find the optimal labeling. Their approach outperformed previous studies on the CleanEval benchmark [3].

Web Complexity. While content extraction has attracted much attention in the scientific literature, fewer studies are conducted to understand website complexity and its impact on page load time and user experience. Gibson *et al.* [14] analyzed webpage template evolution using site-level template detection algorithms and found that templates, with little raw content value, represented 40-50% of the data on the Web and the rate continued to grow at a rate about 6% per year. Butkiewicz *et al.* [7] showed that modern websites, regardless of their popularity, were complex and such complexity could affect user experience. Moreover, their analysis demonstrated that the number of loaded objects and servers could indicate page load time, and both numbers were significant in News websites.

Performance and User Experience. While complexity of webpages can affect page load time, their visual complexity can impact user experience. Harper *et al.*showed that visual complexity in webpages, defined as diversity, density, and positioning of the elements, could increase cognitive load [19] and even have detrimental cognitive and emotional impact on users [41]. In many websites, online advertisements are the only source of income. Nonetheless, online ads, especially intrusive ads, have usability consequences [6]. As Pujol *et al.* [36] observed, 22% of the most active users of a major European ISP use Adblock Plus. As a result, providing the main content in a clutter free page, such as Reader Mode, not only decreases the complexity of a page, but also preserves privacy by limiting the number of requests for third-party services and trackers [12, 25] as well as improves user experience.

7 CONCLUSION

The modern web's progress has led us to the point far beyond Hypertext Markup for document discovery, to having full-fledged, media-rich experiences and dynamic applications. With this growth in capability, there has been a growth in page "bloat", making pages expensive to load, and bringing with it ubiquitous advertising and tracking. In this work, we propose SpeedReader as an approach broadening the applicability of "reader mode" browser features to deliver huge improvements to the end-user browsing experience.

Unique among reader mode tools, SpeedReader determines if a page is readable based only on the page's initial HTML, before the HTML is parsed and rendered, and before sub-resources are fetched. Our classifier can classify within 2 ms and with 91% accuracy, which makes it practical as an always-on part of the rendering pipeline to transform all suitable pages at load time. We find that SpeedReader is widely applicable, and can deliver performance and privacy improvements to 22% of pages on popular and unpopular websites, and a larger proportion of pages linked to from online social networks like Reddit (42%) and Twitter (31%). Since SpeedReader makes its modifications before sub-resources are fetched, it uses 84× less network than traditional page rendering (and current reader mode techniques). This results in page load time improvements, important in a range of scenarios from poor connectivity or low-end devices, to expensive data connectivity or simply wanting a clean and simple interaction with primarily textual content. SpeedReader also delivers page loading speedups of 20× - 27× and average memory reduction of 2.4×, while maintaining a pleasant, reader mode style user experience. Finally, when SpeedReader was applied to 19,765 readable webpages, it prevented 100% of advertising and tracking related resources from being fetched (as labeled by EasyList and EasyPrivacy).

8 ACKNOWLEDGEMENT

This research was supported by Brave Software. We would like to thank David Temkin for his practical feedback and helpful comments on the project. We also would like to thank anonymous reviewers for their time and effort in reviewing this paper.

REFERENCES

[1] Amazon. [n. d.]. Amazon Silk Documentation. docs.aws.amazon.com/silk/index.html

[2] Arc90. [n. d.]. Readability - An Arc90 Lab Experiment. http://ejucovy.github.io/readability/

[3] Marco Baroni, Francis Chantree, Adam Kilgarriff, and Serge Sharoff. 2008. Cleaneval: a Competition for Cleaning Web Pages.. In *LREC*.

[4] Alexander Borisov. [n. d.]. myHTML - Fast C/C++ HTML 5 Parser. Using threads. https://github.com/lexborisov/myhtml

[5] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. 2000. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '00)*. ACM, New York, NY, USA, 297–304. https://doi.org/10.1145/332040.332447

[6] Giorgio Brajnik and Silvia Gabrielli. 2010. A review of online advertising effects on the user experience. *International Journal of Human-Computer Interaction* 26, 10 (2010), 971–997.

[7] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. 2011. Understanding Website Complexity: Measurements, Metrics, and Implications. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC '11)*. ACM, New York, NY, USA, 313–328. https://doi.org/10.1145/2068816.2068846

[8] Deng Cai, Shipeng Yu, Ji-Rong Wen, and Wei-Ying Ma. 2003. VIPS: a Vision-based Page Segmentation Algorithm. (November 2003), 28. https://www.microsoft.com/en-us/research/publication/vips-a-vision-based-page-segmentation-algorithm/

[9] Mozilla Corporation. 2018. Readability.js. https://github.com/mozilla/readability

[10] EasyList. 2018. About EasyList. https://easylist.to/pages/about.html

[11] EasyList. 2018. EasyList Github repository. https://github.com/easylist/easylist

[12] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1388–1401. https://doi.org/10.1145/2976749.2978313

[13] Jian Fan, Ping Luo, Suk Hwan Lim, Sam Liu, Parag Joshi, and Jerry Liu. 2011. Article Clipper: A System for Web Article Extraction. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '11)*. ACM, New York, NY, USA, 743–746. https://doi.org/10.1145/2020408.2020525

[14] David Gibson, Kunal Punera, and Andrew Tomkins. 2005. The Volume and Evolution of Web Page Templates. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web (WWW '05)*. ACM, New York, NY, USA, 830–839. https://doi.org/10.1145/1062745.1062763

[15] Eyeo GmbH. 2018. Adblock Plus. https://adblockplus.org/

[16] Utkarsh Goel, Moritz Steiner, Mike P Wittie, Martin Flack, and Stephen Ludin. 2017. Measuring What is Not Ours: A Tale of 3rd Party Performance. In *International Conference on Passive and Active Network Measurement*. Springer, 142–155.

[17] Google. [n. d.]. Accelerated Mobile Pages Project. https://www.ampproject.org

[18] Suhit Gupta, Gail Kaiser, David Neistadt, and Peter Grimm. 2003. DOM-based Content Extraction of HTML Documents. In *Proceedings of the 12th International Conference on World Wide Web (WWW '03)*. ACM, New York, NY, USA, 207–214. https://doi.org/10.1145/775152.775182

[19] Simon Harper, Eleni Michailidou, and Robert Stevens. 2009. Toward a Definition of Visual Complexity As an Implicit Measure of Cognitive Load. *ACM Trans. Appl. Percept.* 6, 2, Article 10 (March 2009), 18 pages. https://doi.org/10.1145/1498700.1498704

[20] Raymond Hill. 2018. uBlock Origin - An efficient blocker for Chromium and Firefox. Fast and lean. https://github.com/gorhill/uBlock

[21] Brave Software Inc. 2018. Brave Ad Block. https://github.com/brave/ad-block

[22] Google Inc. [n. d.]. Catapult - Web Page Replay. https://github.com/catapult-project/catapult.git

[23] Google Inc. 2018. DOM Distiller. https://github.com/chromium/dom-distiller

[24] Christian Kohlschütter, Peter Fankhauser, and Wolfgang Nejdl. 2010. Boilerplate Detection Using Shallow Text Features. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining (WSDM '10)*. ACM, New York, NY, USA, 441–450. https://doi.org/10.1145/1718487.1718542

[25] Balachander Krishnamurthy and Craig Wills. 2009. Privacy Diffusion on the Web: A Longitudinal Perspective. In *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*. ACM, New York, NY, USA, 541–550. https://doi.org/10.1145/1526709.1526782

[26] Deepak Kumar, Zane Ma, Zakir Durumeric, Ariana Mirian, Joshua Mason, J. Alex Halderman, and Michael Bailey. 2017. Security Challenges in an Increasingly Tangled Web. In *Proceedings of the 26th International Conference on World Wide Web (WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 677–684. https://doi.org/10.1145/3038912.3052686

[27] Eduardo Sany Laber, Críston Pereira de Souza, Iam Vita Jabour, Evelin Carvalho Freire de Amorim, Eduardo Teixeira Cardoso, Raúl Pierre Rentería, Lúcio Cunha Tinoco, and Caio Dias Valentim. 2009. A Fast and Simple Method for Extracting Relevant Content from News Webpages. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM '09)*. ACM, New York, NY, USA, 1685–1688. https://doi.org/10.1145/1645953.1646204

[28] Timothy Libert. 2015. Exposing the Invisible Web: An Analysis of Third-Party HTTP Requests on 1 Million Websites. *International Journal of Communication* 9, 0 (2015). https://ijoc.org/index.php/ijoc/article/view/3646

[29] Shian-Hua Lin and Jan-Ming Ho. 2002. Discovering Informative Content Blocks from Web Documents. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '02)*. ACM, New York, NY, USA, 588–593. https://doi.org/10.1145/775047.775134

[30] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. 2017. Block me if you can: A large-scale study of tracker-blocking tools. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*. IEEE, 319–333.

[31] Georg Merzdovnik, Markus Huber, Damjan Buhov, Nick Nikiforakis, Sebastian Neuner, Martin Schmiedecker, and Edgar Weippl. 2017. Block Me if You Can: A Large-Scale Study of Tracker-Blocking Tools. *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017* (2017), 319–333. https://doi.org/10.1109/EuroSP.2017.26

[32] mikesizz. [n. d.]. RedditList - Tracking the top 5000 subreddits. http://redditlist.com/

[33] Thomas Nagele. 2015. *Client-side performance profiling of JavaScript for web applications*. Master Thesis. Radboud University Nijmegen.

[34] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2012. You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 736–747. https://doi.org/10.1145/2382196.2382274

[35] Jeff Pasternack and Dan Roth. 2009. Extracting Article Text from the Web with Maximum Subsequence Segmentation. In *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*. ACM, New York, NY, USA, 971–980. https://doi.org/10.1145/1526709.1526840

[36] Enric Pujol, Oliver Hohlfeld, and Anja Feldmann. 2015. Annoyed Users: Ads and Ad-Block Usage in the Wild. In *Proceedings of the 2015 Internet Measurement Conference (IMC '15)*. ACM, New York, NY, USA, 93–106. https://doi.org/10.1145/2815675.2815705

[37] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps'10)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1863166.1863169

[38] Borut Sluban and Miha Grčar. 2013. URL tree: efficient unsupervised content extraction from streams of web documents. In *Proceedings of the 22nd ACM international conference on Conference on information &#38; knowledge management (CIKM '13)*. ACM, New York, NY, USA, 2267–2272. https://doi.org/10.1145/2505515.2505654

[39] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. 2016. Browser Feature Usage on the Modern Web. In *Proceedings of the 2016 Internet Measurement Conference (IMC '16)*. ACM, New York, NY, USA, 97–110. https://doi.org/10.1145/2987443.2987466

[40] Fei Sun, Dandan Song, and Lejian Liao. 2011. DOM Based Content Extraction via Text Density. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '11)*. ACM, New York, NY, USA, 245–254. https://doi.org/10.1145/2009916.2009952

[41] Alexandre N. Tuch, Javier A. Bargas-Avila, Klaus Opwis, and Frank H. Wilhelm. 2009. Visual complexity of websites: Effects on users' experience, physiology, performance, and memory. *International Journal of Human-Computer Studies* 67, 9 (2009), 703 – 715. https://doi.org/10.1016/j.ijhcs.2009.04.002

[42] Antoine Vastel, Peter Snyder, and Benjamin Livshits. 2018. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. (2018). http://arxiv.org/abs/1810.09160

[43] Thijs Vogels, Octavian-Eugen Ganea, and Carsten Eickhoff. 2018. Web2Text: Deep Structured Boilerplate Removal. In *European Conference on Information Retrieval*. Springer, 167–179.

[44] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 473–485. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/wang_xiao

[45] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up Web Page Loads with Shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 109–122. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/wang

[46] Zhiheng Wang. 2012. *Navigation Timing*. W3C Recommendation. W3C. http://www.w3.org/TR/2012/REC-navigation-timing-20121217/.

[47] Tim Weninger, William H. Hsu, and Jiawei Han. 2010. CETR: Content Extraction via Tag Ratios. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 971–980. https://doi.org/10.1145/1772690.1772789

[48] Shanchan Wu, Jerry Liu, and Jian Fan. 2015. Automatic Web Content Extraction by Combination of Learning and Grouping. In *Proceedings of the 24th International Conference on World Wide Web (WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1264–1274. https://doi.org/10.1145/2736277.2741659